8.1 Why pointers: Pass by pointer example

A challenging but powerful programming construct is something called a *pointer*. This section illustrates one example's beneficial usage of pointers, namely pass by pointer function parameters.

A function can only return one value. But consider a desired function that converts total inches into feet and inches, e.g., 95 inches would be converted to 7 feet and 11 inches. To effectively return two values, the function can be defined with two ***pass by pointer*** parameters, by putting a * before a parameter name, and & before the corresponding argument variable[parm].

The & passes the variable's memory address, known as a ***pointer***, rather than the variable's value. The * before the parameter name indicates the parameter is a pointer. The function's statements can update each argument variable's memory location, effectively "returning" a value. The technique is also known as ***pass by reference***, but the term pass by pointer avoids confusion with pass by reference parameters in C++ programs (which are different), and to more accurately describe this technique.

The following animation illustrates how pass by value does not work to return two values from a function.

PARTICIPATION ACTIVITY 8.1.1: Pointer example: Without pass by pointer parameters.

Animation captions:

1. ConvFeetInches' parameters are passed by value, so the arguments' values are copied into local variables.
2. Upon return, ConvFeetInches' are discarded so the function fails to update the resFeet and resIn variables.

The following animation illustrates how pass by pointer effectively enables a function to return two values.

PARTICIPATION ACTIVITY 8.1.2: Pointer example: Pass by pointer parameters.

Animation captions:

1. The & before the argument indicates that a variable's memory addresses, known as a pointer, is passed to a pass-by-pointer parameter. The * before the parameter name indicates the parameter is a pointer.
2. Prepending "*" to a pointer variable's name access the value pointed to by the pointer, so the original variable is updated.

3. Upon return from ConvFeetInches, resFeet and resIn retain their updated values, effectively "returning" two values.

Pass by pointer parameters should be used only when the output values are tightly related. New programmers commonly create one function with two outputs (using pass by pointer) to reduce coding, where two functions would have been better. For example, defining two functions `int StepsToFeet(int baseSteps)` and `int StepsToCalories(int baseCalories)` is better than defining a single function `void StepsToFeetAndCalories(int baseSteps, int* feetTot, int* caloriesTot)`. Defining separate functions supports modular development, and enables use of the functions in an expression as in `if (StepsToFeet(baseSteps) < 100)`.

Good candidates for multiple pass by pointer parameters might include computing the number of each type of coin to give as change, whose function might be `void ComputeChange(int totCents, int* numQuarters, int* numDimes, int* numNickels`. Another example is converting from polar coordinates to Cartesian coordinates, whose function might be `void PolarToCartesian(int radialPol, int anglePol, int* xCar, int* yCar)`. In both situations, the two outputs are tightly related.

zyDE 8.1.1: Calculating change.

Complete the program to compute the number of quarter, dime, nickel, and penny coins equal total cents, using the fewest coins (i.e., using the most possible of a larger coin fi

Load default template... Run

```
1
2  #include <stdio.h>
3
4  void ComputeChange(int totCents) { /
5     printf("FIXME: Finish this functi
6  }
7
8  int main(void) {
9
10    int totalCents;         // Total
11    int quartersChange;     // Number
12    int dimesChange;        // Number
13    int nickelsChange;      // Number
14    int penniesChange;      // Number
15
16    totalCents = 67;
17
18
```

PARTICIPATION ACTIVITY 8.1.3: Why pointer.

1) Complete the function declaration for MyFct with input parameters w and x, and "output" parameters y and z, in that order, all dealing with type int.

```
void MyFct(
);
```

Check Show answer

2) Call a function MyFct that returns void, with four parameters, the last two being pointers. Call the function with argument variables a, b, c, and d, in that order, all being of type int.

Check Show answer

Exploring further:

- Pointers tutorial from msdn.microsoft.com

CHALLENGE ACTIVITY 8.1.1: Calling a function that has pass by pointer parameters.

Write a function call with arguments tensPlace, onesPlace, and userInt. Be sure to pass the first two arguments as pointers. Sample output for the given program:


```
tensPlace = 4, onesPlace = 1
```

347282.2052980.qx3zqy7

```
1 #include <stdio.h>
2
3 void SplitIntoTensOnes(int* tensDigit, int* onesDigit, int DecVal){
```

```
4     *tensDigit = (DecVal / 10) % 10;
5     *onesDigit = DecVal % 10;
6  }
7
8  int main(void) {
9     int tensPlace;
10    int onesPlace;
11    int userInt;
12
13    scanf("%d", &userInt);
14
15    /* Your solution goes here  */
16
17    printf("tensPlace = %d, onesPlace = %d\n", tensPlace, onesPlace);
18
```


Run

View your last submission

CHALLENGE ACTIVITY 8.1.2: Pass by pointer: Adjusting start/end times.

Define a function UpdateTimeWindow() with parameters timeStart, timeEnd, and offsetAmount. Each parameter is of type int. The function adds offsetAmount to each of the first two parameters. Make the first two parameters pass by pointer. Sample output for the given program:

```
timeStart = 3, timeEnd = 7
timeStart = 5, timeEnd = 9
```

347282.2052980.qx3zqy7

```
1  #include <stdio.h>
2
3  // Define void UpdateTimeWindow(...)
4
5  /* Your solution goes here  */
6
7  int main(void) {
8     int timeStart;
9     int timeEnd;
10    int offsetAmount;
11
12    scanf("%d", &timeStart);
13    scanf("%d", &timeEnd);
14    scanf("%d", &offsetAmount);
15    printf("timeStart = %d, timeEnd = %d\n", timeStart, timeEnd);
16
17    UpdateTimeWindow(&timeStart, &timeEnd, offsetAmount);
```

```
18    printf( timeStart = %d, timeEnd = %d\n , timeStart, timeEnd);
```

Run

View your last submission

(*parm) Recall that the *parameter* is part of the function definition, while the *argument* is the item passed in a function call

8.2 Pointer basics

Pointer variables

A ***pointer*** is a variable that holds a memory address, rather than holding data like most variables. A pointer has a data type, and the data type determines what type of address is held in the pointer. Ex: An integer pointer holds a memory address of an integer, and a double pointer holds an address of a double. A pointer is declared by including * before the pointer's name. Ex: `int* maxItemPointer` declares an integer pointer named maxItemPointer.

Typically, a pointer is initialized with another variable's address. The ***reference operator*** (&) obtains a variable's address. Ex: `&someVar` returns the memory address of variable someVar. When a pointer is initialized with another variable's address, the pointer "points to" the variable.

PARTICIPATION ACTIVITY 8.2.1: Assigning a pointer with an address.

Animation content:

`undefined`

Animation captions:

1. someInt is located in memory at address 76.
2. valPointer is located in memory at address 78. valPointer has not been initialized, so valPointer points to an unknown address.
3. someInt is assigned with 5. The reference operator & returns someInt's address 76.
4. valPointer is assigned with the memory address of someInt, so valPointer points to someInt.

Printing memory addresses

The example above uses the printf() format specifier %p to output memory addresses. %p expects the data type void, but an int variable's memory address is int*. So `&someInt` and `valPointer` are type cast to the data type void*. %p outputs a memory address using hexadecimal numbers like 0x7ffc3ae4f0e4, which is a base 16 number. The animation above outputs memory addresses as decimal numbers like 76 for simplicity.*

PARTICIPATION ACTIVITY 8.2.2: Declaring and initializing a pointer.

1) Declare a double pointer called sensorPointer.

Check Show answer

2) Output the address of the double variable sensorVal.

```
printf("%p", (void*)
            );
```

Check Show answer

3) Assign sensorPointer with the variable sensorVal's address. In other words, make sensorPointer point to sensorVal.

Check Show answer

Dereferencing a pointer

The ***dereference operator*** (*) is prepended to a pointer variable's name to retrieve the data to which the pointer variable points. Ex: If valPointer points to a memory address containing the integer 123, then `printf("%d", *valPointer);` dereferences valPointer and outputs 123.

PARTICIPATION

ACTIVITY 8.2.3: Using the dereference operator.

Animation content:

undefined

Animation captions:

1. someInt is located in memory at address 76, and valPointer points to someInt.
2. The dereference operator * gets the value pointed to by valPointer, which is 5.
3. Assigning *valPointer with a new value changes the value valPointer points to. The 5 changes to 10.
4. Changing *valPointer also changes someInt. someInt is now 10.

PARTICIPATION ACTIVITY 8.2.4: Dereferencing a pointer.

Refer to the code below.

```
char userLetter = 'B';
char* letterPointer;
```

1) What line of code makes letterPointer point to userLetter?

- `letterPointer = userLetter;`
- `*letterPointer = &userLetter;`
- `letterPointer = &userLetter;`

2) What line of code assigns the variable outputLetter with the value letterPointer points to?

- `outputLetter = letterPointer;`
- `outputLetter = *letterPointer;`
- `someChar = &letterPointer;`

3) What does the code output?

```
letterPointer = &userLetter;
userLetter = 'A';
*letterPointer = 'C';
printf("%c", userLetter);
```

- A
- B
- C

Null pointer

When a pointer is declared, the pointer variable holds an unknown address until the pointer is initialized. A programmer may wish to indicate that a pointer points to "nothing" by initializing a pointer to null. ***Null*** means "nothing". A pointer that is assigned with the macro ***NULL*** is said to be null. NULL is defined as 0 on most systems because 0 is not a valid memory address. Ex:
`int *maxValPointer = NULL;` makes maxValPointer null.

In the animation below, the function PrintValue() only outputs the value pointed to by valuePointer if valuePointer is not null.

PARTICIPATION ACTIVITY 8.2.5: Checking to see if a pointer is null.

Animation content:

undefined

Animation captions:

1. someInt is located in memory at address 76. valPointer is assigned with NULL.
2. valPointer is passed to PrintValue(), so the valuePointer parameter is also NULL.
3. The if statement is true since valuePointer is NULL.
4. valPointer points to someInt, so calling PrintValue() assigns valuePointer with the address 76.
5. The if statement is false because valuePointer is no longer NULL. valuePointer points to the value 5, so 5 is output.

PARTICIPATION ACTIVITY 8.2.6: Null pointer.

Refer to the animation above.

1) The code below outputs 3.

```
int numSides = 3;
int* valPointer = &numSides;
PrintValue(valPointer);
```

- True
- False

2) The code below outputs 5.


```
int numSides = 5;
int* valPointer = &numSides;
valPointer = NULL;
PrintValue(valPointer);
```

- True
- False

3) The code below outputs 7.

```
int numSides = 7;
int* valPointer = NULL;
printf("%d", *valPointer);
```

- True
- False

Common pointer errors

A number of common pointer errors result in syntax errors or warnings that are caught by the compiler or runtime errors that may result in the program crashing.

Common syntax errors:

- A common error is to use the dereference operator when initializing a pointer. Ex: `*valPointer = &maxValue;` is a syntax error because *valPointer is referring to the value pointed to, not the pointer itself.
- A common error when declaring multiple pointers on the same line is to forget the * before each pointer name. Ex: `int* valPointer1, valPointer2;` declares valPointer1 as a pointer, but valPointer2 is declared as an integer because no * exists before valPointer2. Good practice is to declare one pointer per line to avoid accidentally declaring a pointer incorrectly.

Common runtime errors:

- A common error is to use the dereference operator when a pointer has not been initialized. Ex: `printf("%d", *valPointer);` may cause a program to crash if valPointer holds an unknown address or an address the program is not allowed to access.
- A common error is to dereference a null pointer. Ex: If valPointer is null, then `printf("%d", *valPointer);` causes the program to crash. A pointer should always hold a

valid address before the pointer is dereferenced.

PARTICIPATION ACTIVITY 8.2.7: Common pointer errors.

Animation content:

undefined

Animation captions:

1. A syntax error results if valPointer is assigned using the dereference operator *.
2. Multiple pointers cannot be declared on a single line with only one asterisk. Good practice is to declare each pointer on a separate line.
3. valPointer is not initialized, so valPointer contains an unknown address. Dereferencing an unknown address may cause a runtime error.
4. valPointer is null, and dereferencing a null pointer causes a runtime error.

PARTICIPATION ACTIVITY 8.2.8: Common pointer errors.

Indicate if each code segment has a syntax error, runtime error, or no error.

1)

```
char* newPointer;
*newPointer = 'A';
printf("%d", *newPointer);
```

- syntax error
- runtime error
- no errors

2)

```
char* valPointer1, *valPointer2;
valPointer1 = NULL;
valPointer2 = NULL;
```

- syntax error
- runtime error
- no errors

3)

```
char someChar = 'Z';
char* valPointer;
*valPointer = &someChar;
```

- syntax error

- runtime error
- no errors

4)

```
char* newPointer = NULL;
char someChar = 'A';
*newPointer = 'B';
```

- syntax error
- runtime error
- no errors

Two pointer declaration styles

Some programmers prefer to place the asterisk next to the variable name when declaring a pointer. Ex: ***`int *valPointer;`****. The style preference is useful when declaring multiple pointers on the same line:* ***`int *valPointer1, *valPointer2;`****.*
Good practice is to use the same pointer declaration style throughout the code: Either ***`int* valPointer`*** *or* ***`int *valPointer`****.*

This material uses the style ***`int* valPointer`*** *and always declares one pointer per line to avoid accidentally declaring a pointer incorrectly.*

Advanced compilers can check for common errors

Some compilers have advanced code analysis capabilities to catch some runtime errors at compile time. Ex: The compiler may issue a warning if the compiler detects a null pointer is being dereferenced. An advanced compiler can never catch all runtime errors because a potential runtime error may depend on user input, which is unknown at compile time.

zyDE 8.2.1: Using pointers.

The following provides an example (not useful other than for learning) of assigning the address of variable vehicleMpg to the pointer variable valPointer.

1. Run and observe that the two output statements produce the same output.

2. Modify the value assigned to *valPointer and run again.
3. Now uncomment the statement that assigns vehicleMpg. PREDICT whether both statements will print the same output. Then run and observe the output. Did you p correctly?

Load default template...

Run

```
1
2  #include <stdio.h>
3
4  int main(void) {
5     double vehicleMpg;
6     double* valPointer = NULL;
7
8     valPointer = &vehicleMpg;
9
10    *valPointer = 29.6; // Assigns th
11                        // POINTED TO
12
13   // vehicleMpg = 40.0;    // Uncomme
14
15    printf("Vehicle MPG = %lf\n", veh
16    printf("Vehicle MPG = %lf\n", *va
17    return 0;
18 }
```


CHALLENGE ACTIVITY 8.2.1: Enter the output of pointer content.

347282.2052980.qx3zqy7

Start

Type the program's output

```
#include <stdio.h>

int main(void) {
   int someNumber;
   int* numberPointer;

   someNumber = 8;
   numberPointer = &someNumber;

   printf("%d %d\n", someNumber, *numberPointer);

   return 0;
}
```


1 2

Check Next

CHALLENGE ACTIVITY 8.2.2: Printing with pointers.

If the input is negative, make numItemsPointer be null. Otherwise, make numItemsPointer point to numItems and multiply the value to which numItemsPointer points by 10. Ex: If the user enters 99, the output should be:

```
Items: 990
```

347282.2052980.qx3zqy7

```
#include <stdio.h>

int main(void) {
   int* numItemsPointer;
   int numItems;

   scanf("%d", &numItems);

   /* Your solution goes here  */

   if (numItemsPointer == NULL) {
      printf("Items is negative\n");
   }
   else {
      printf("Items: %d\n", *numItemsPointer);
   }

   return 0;
```

Run

View your last submission

8.3 The malloc and free functions

Sometimes memory should be allocated while a program is running and should persist independently of any particular function. The malloc() function carries out such memory allocation.

malloc() is a function defined within the ***standard library***, which can be used by adding `#include <stdlib.h>` to the beginning of a program. The malloc() function is named for *m*emory *allo*cation. malloc() allocates memory of a given number of bytes and returns a pointer (i.e., the address) to that allocated memory. A basic call to malloc() has the form:

Construct 8.3.1: Malloc function.

```
malloc(bytes)
```


malloc() takes a single argument specifying the number of bytes to allocate in memory. Thus, the programmer must determine the number of bytes needed to allocate space for the desired data type. But, as you may recall, the number of bytes for various data types (e.g., int) may vary across different computer systems. Fortunately, C provides a sizeof() operator that returns the number of bytes for a given data type. For example, on a system where an int is 32 bits, `sizeof(int)` returns 4, because 32-bits is 4 bytes. Calls to malloc() are typically combined with sizeof() using the form:

Construct 8.3.2: Malloc and sizeof functions.

```
malloc(sizeof(type))
```

malloc() returns a pointer to allocated memory using a `void*`, referred to as a ***void pointer***. A void pointer is a special type of pointer that stores a memory address without referring to the type of variable stored at that memory location. The void pointer return type allows a single malloc() function to allocate memory for any data type. The pointer returned from malloc() can then be written into a particular pointer variable by casting the void pointer to the data type using the form:

Construct 8.3.3: Malloc return type.

```
pointerVariableName = (type*)malloc(sizeof(type));
```


The following animation illustrates using malloc() to allocate memory for an int. int is used for introduction; malloc() is more commonly used to allocate memory for struct types, arrays, and strings.

PARTICIPATION ACTIVITY 8.3.1: The malloc() operation.

Animation captions:

1. Variable myPtr is a pointer of type int, which does not yet point to anything.
2. malloc(sizeof(int)) allocates memory for a single int. The void pointer returned by malloc is first cast to an int pointer before myPtr is assigned with that pointer.
3. Dereferencing the pointer variable allows the program to assign and access the value pointed to by the pointer.

The malloc() function returns NULL if the function failed to allocate memory. Such failure could happen if a program has used up all memory available to the program.

free() is a function that deallocates a memory block pointed to by a given pointer, which must have been previously allocated by malloc(). In other words, free() does the opposite of malloc(). Deallocating memory using free() has the following form:

Construct 8.3.4: The free function.

```
free(pointerVariable);
```

After `free(pointerVariable);`, the program should not attempt to dereference pointerVariable, as pointerVariable points to a memory location that is no longer allocated for use by pointerVariable. Dereferencing a pointer whose memory has been deallocated is a common error, and may cause strange program behavior that is difficult to debug — if that memory had since been allocated to another variable, that variable's value could mysteriously change. Calling free with a pointer that wasn't previously allocated by malloc is also an error.

PARTICIPATION ACTIVITY 8.3.2: malloc and free.

1) Declare a variable named myValPointer as a pointer of type int, initializing the pointer to NULL in the declaration.

Check Show answer

2) Write a statement that allocates memory for a new double value

using the pointer variable newInputPtr.

Check Show answer

3) Write a statement that deallocates memory for the pointer variable newInputPtr.

Check Show answer

Exploring further:

- malloc Reference Page from cplusplus.com
- More on malloc from msdn.microsoft.com
- free Reference Page from cplusplus.com
- More on free from msdn.microsoft.com

CHALLENGE ACTIVITY 8.3.1: Using malloc and pointers.

Write two statements that each use malloc to allocate an int location for each pointer.
Sample output for given program:

```
numPtr1 = 44, numPtr2 = 99
```

347282.2052980.qx3zqy7


```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void) {
5     int* numPtr1 = NULL;
6     int* numPtr2 = NULL;
7
8     /* Your solution goes here  */
9
10    scanf("%d", numPtr1);
```

```
11    scanf("%d", numPtr2);
12
13    printf("numPtr1 = %d, numPtr2 = %d\n", *numPtr1, *numPtr2);
14
15    free(numPtr1);
16    free(numPtr2);
17
18    return 0;
```

Run

View your last submission

8.4 String functions with pointers

C string library functions

The C string library, introduced elsewhere, contains several functions for working with C strings. This section describes the use of char pointers in such functions. The C string library must first be included via: `#include <string.h>`.

Each string library function operates on one or more strings, each passed as a char* or const char* argument. Strings passed as char* can be modified by the function, whereas strings passed as const char* arguments cannot. Examples of such functions are strcmp() and strcpy(), introduced elsewhere.

PARTICIPATION ACTIVITY 8.4.1: strcmp() and strcpy() string functions.

Animation content:

undefined

Animation captions:

1. strcmp() compares 2 strings. Since neither string is modified during the comparison, each parameter is a const char*.
2. strcmp() returns an integer that is 0 if the strings are equal, non-zero if the strings are not equal.
3. strcpy() copies a source string to a destination string. The destination string gets modified and thus is a char*.
4. The source string is not modified and thus is a const char*.

5. strcpy() copies 4 characters, "xyz" and the null-terminator, to newText.

PARTICIPATION ACTIVITY 8.4.2: C string library functions.

1) A variable declared as `char* substringAt5 = &myString[5];` cannot be passed as an argument to strcmp(), since strcmp() requires const char* arguments.
   - True
   - False

2) A character array variable declared as `char myString[50];` can be passed as either argument to strcpy().
   - True
   - False

3) A variable declared as `const char* firstMonth = "January";` could be passed as either argument to strcpy().
   - True
   - False

C string search functions

strchr(), strrchr(), and strstr() are C string library functions that search strings for an occurrence of a character or substring. Each function's first parameter is a const char*, representing the string to search within.

The strchr() and strrchr() functions find a character within a string, and thus have a char as the second parameter. strchr() finds the first occurrence of the character within the string and strrchr() finds the last occurrence.

strstr() searches for a substring within another string, and thus has a const char* as the second parameter.

Table 8.4.1: Some C string search functions.

Given:

```
char orgName[100] = "The Dept. of Redundancy Dept.";
char newText[100];
char* subString = NULL;
```

strchr()	`strchr(sourceStr, searchChar)`
	
	Returns NULL if searchChar does not exist in sourceStr. Else, returns pointer to first occurrence.
		`if (strchr(orgName, 'D') != NULL) {`
		`// 'D' exists in orgName?`
		`  subString = strchr(orgName, 'D');`
		`// Points to first 'D'`
		`  strcpy(newText, subString);`
		`// newText now "Dept. of Redundancy Dept."`
		`}`
		`if (strchr(orgName, 'Z') != NULL) {`
		`// 'Z' exists in orgName?`
		`  ... // Doesn't exist, branch not taken`
		`}`

strrchr()	`strrchr(sourceStr, searchChar)`
	
	Returns NULL if searchChar does not exist in sourceStr. Else, returns pointer to LAST occurrence (searches in reverse, hence middle 'r' in name).
		`if (strrchr(orgName, 'D') != NULL) {`
		`// 'D' exists in orgName?`
		`  subString = strrchr(orgName, 'D');`
		`// Points to last 'D'`
		`  strcpy(newText, subString);`
		`// newText now "Dept."`
		`}`

strstr()	`strstr(str1, str2)`
	
	Returns NULL if str2 does not exist in str1. Else, returns a char pointer pointing to first occurrence of string str2 within string str1.
		`subString = strstr(orgName, "Dept");`
		`// Points to first 'D'`
		`if (subString != NULL) {`
		`  strcpy(newText, subString);`
		`// newText now "Dept. of Redundancy Dept."`
		`}`

PARTICIPATION ACTIVITY 8.4.3: C string search functions.

1) What does fileExtension point to after the following code?

```
const char* fileName =
"Sample.file.name.txt";
const char* fileExtension =
strrchr(fileName, '.');
```


- ".file.name.txt"
- ".txt"
- "Sample.file.name"

2) `strstr(fileName, ".pdf")` is non-null only if the fileName string ends

with ".pdf".

- True
- False

3) What is true about fileName if the following expression evaluates to true?

```
strchr(fileName, '.') ==
strrchr(fileName, '.')
```

- The '.' character occurs exactly once in fileName.
- The '.' character occurs 0 or 1 time in fileName.
- The '.' character occurs 1 or more times in fileName.

Search and replace example

The following example carries out a simple censoring program, replacing an exclamation point by a period and "Boo" by "---" (assuming those items are somehow bad and should be censored.) "Boo" is replaced using the strncpy() function, which is described elsewhere.

Note that only the first occurrence of "Boo" is replaced, as strstr() returns a pointer to the first occurrence. Additional code would be needed to delete all occurrences.

Figure 8.4.1: String searching example.

```
Enter a line of text: Hello!
Censored: Hello.

...

Enter a line of text: Boo hoo to you!
Censored: --- hoo to you.

...

Enter a line of text: Booo! Boooo!!!!
Censored: ---o. Boooo!!!!
```

```c
#include <stdio.h>
#include <string.h>

int main(void) {
   const int MAX_USER_INPUT = 100;  // Max input size
   char userInput[MAX_USER_INPUT];  // User defined string
   char* stringPos = NULL;          // Index into string

   // Prompt user for input
   printf("Enter a line of text: ");
   fgets(userInput, MAX_USER_INPUT, stdin);

   // Locate exclamation point, replace with period
   stringPos = strchr(userInput, '!');

   if (stringPos != NULL) {
      *stringPos = '.';
   }

   // Locate "Boo" replace with "---"
   stringPos = strstr(userInput, "Boo");

   if (stringPos != NULL) {
      strncpy(stringPos, "---", 3);
   }

   // Output modified string
   printf("Censored: %s\n", userInput);

   return 0;
}
```


PARTICIPATION ACTIVITY 8.4.4: Modifying and searching strings.

1) Declare a char* variable named charPtr.

Check Show answer

2) Assuming `char* firstR;` is already declared, store in firstR a pointer to the first instance of an 'r' in the char* variable userInput.

Check Show answer

3) Assuming `char* lastR;` is already

declared, store in lastR a pointer to the last instance of an 'r' in the char* variable userInput.

Check Show answer

4) Assuming `char* firstQuit;` is already declared, store in firstQuit a pointer to the first instance of "quit" in the char* variable userInput.

Check Show answer

CHALLENGE ACTIVITY 8.4.1: Enter the output of the String functions.

347282.2052980.qx3zqy7

Start

Type the program's output

```
#include <stdio.h>
#include <string.h>

int main(void) {
   char nameAndTitle[50];
   char subString[20];
   char* stringPointer = NULL;

   strcpy(nameAndTitle, "Ms. Ann Hill");

   stringPointer = strchr(nameAndTitle, 'H');
   strcpy(subString, stringPointer);

   printf("%s\n", subString);

   return 0;
}
```


1 2 3

Check Next

CHALLENGE ACTIVITY 8.4.2: Find char in string.

Assign searchResult with a pointer to any instance of searchChar in personName. Hint: Use strchr().

347282.2052980.qx3zqy7


```
#include <stdio.h>
#include <string.h>

int main(void) {
   char personName[100];
   char searchChar;
   char* searchResult = NULL;

   fgets(personName, 100, stdin);
   scanf("%c", &searchChar);

   /* Your solution goes here  */

   if (searchResult != NULL) {
      printf("Character found.\n");
   }
   else {
      printf("Character not found.\n");
```

Run

View your last submission

CHALLENGE ACTIVITY 8.4.3: Find string in string.

Assign movieResult with the first instance of The in movieTitle.

347282.2052980.qx3zqy7


```
#include <stdio.h>
#include <string.h>

int main(void) {
   char movieTitle[100];
   char* movieResult = NULL;

   fgets(movieTitle, 100, stdin);

   /* Your solution goes here  */

   printf("Movie title contains The? ");
   if (movieResult != NULL) {
      printf("Yes.\n");
   }
```

```
16    else {
17       printf("No.\n");
18    }
```

Run

View your last submission

8.5 The malloc function for arrays and strings

Pointers are commonly used to allocate arrays just large enough to store a required number of data elements. Previously, the programmer had to declare arrays to have a fixed size, referred to as a ***statically allocated*** array. Statically allocated arrays may not use all the allocated memory due to the programmer not knowing the actual needed size before the program runs, thus creating larger-than-necessary arrays. Even then, the program might need a larger size.

Instead of statically allocating an array, malloc() can be used to allocate just enough memory for the array. Recall that arrays are stored in sequential memory locations. malloc() can be used to allocate a ***dynamically allocated*** array by determining the total number of bytes needed to store the desired number of elements, using the following form:

Construct 8.5.1: Dynamically allocated array.

```
pointerVariableName = (dataType*)malloc(numElements * sizeof(dataType))
```

When allocating an array, malloc() returns a pointer to memory location of the first element within the array. This memory location can be stored within a pointer variable declared as a pointer to the type of element within the array. For example, when dynamically allocating an array of integers, a pointer variable of integers "int*" is used. Notice then that an int* pointer can point to either a single integer or an array of multiple characters. A programmer must carefully keep track of how each pointer variable is utilized within the program.

The following program illustrates how to dynamically allocate an arrays of integers.

Figure 8.5.1: Dynamically allocating an array of integers.

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
   int* userVals = NULL; // No array yet
   int numVals;
   int i;

   printf("Enter number of integer values: ");
   scanf("%d", &numVals);

   userVals = (int*)malloc(numVals * sizeof(int));

   printf("Enter %d integer values...\n", numVals);
   for (i = 0; i < numVals; ++i) {
      printf("Value: ");
      scanf("%d", &(userVals[i]));
   }

   printf("You entered: ");
   for (i = 0; i < numVals; ++i) {
      printf("%d ", userVals[i]);
   }
   printf("\n");

   free(userVals);

   return 0;
}
```

```
Enter number of integer values: 8
Enter 8 integer values...
Value: 4
Value: 23
Value: 14
Value: 5
Value: 4
Value: 3
Value: 7
Value: 9
You entered: 4 23 14 5 4 3 7 9
```


PARTICIPATION ACTIVITY 8.5.1: Using malloc for arrays.

1) Write a malloc function call to allocate an array for 10 int variables.

```c
int* itemList = NULL;
itemList = (int*)malloc(10 *
sizeof(          ));
```

Check Show answer

2) Write a malloc function call to allocate an array for 15 double variables.

```c
double* priceList = NULL;
priceList = (
)malloc(15 * sizeof(double));
```

Check Show answer

3) Using malloc, write a statement that allocates an array of 10 chars.

```
char* myStr = NULL;
myStr = (char*)malloc(
                      );
```

Check Show answer

The following program creates a string for a simple greeting given a user entered name.

Figure 8.5.2: Concatenating strings using a statically allocated array.

```
#include <stdio.h>
#include <string.h>

int main(void) {
   char nameArr[100];            // User specified name
   char greetingArr[100];        // Output greeting and name

   // Prompt user to enter a name
   printf("Enter name: ");
   fgets(nameArr, 100, stdin);

   // Eliminate end-of-line char
   nameArr[strlen(nameArr)-1] = '\0';

   // Modify string, hello + user specified name
   strcpy(greetingArr, "Hello ");
   strcat(greetingArr, nameArr);
   strcat(greetingArr, ".");

   // Output greeting and name
   printf("%s\n", greetingArr);

   return 0;
}
```

```
Enter name: Bill
Hello Bill.
```

The above program declares two statically allocated arrays named nameArr and greetingArr. Each array can store a string with 0 to 99 characters — keeping in mind the space needed to store the null character at the end of the string. However, if the user enters the name "Bob", the resulting string stored within the greeting array only requires 11 characters — 6 characters for "Hello ", 3 characters for the name "Bob", 1 character for the ".", and 1 character for the null character. Thus, 88 characters are unused in the greeting array. Likewise, the program fails if the user enters a very long name.

The following program revises the earlier example by dynamically allocating a greeting array to be just large enough to store the entire greeting.

Figure 8.5.3: Concatenating strings using a dynamically allocated array

Figure 8.5.3: Concatenating strings using a dynamically allocated array.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
   char nameArr[100];        // User specified name
   char* greetingPtr = NULL; // Pointer to output greeting and name

   // Prompt user to enter a name
   printf("Enter name: ");
   fgets(nameArr, 100, stdin);

   // Eliminate end-of-line char
   nameArr[strlen(nameArr) - 1] = '\0';

   // Dynamically allocate greeting
   greetingPtr = (char*)malloc((strlen(nameArr) + 8) * sizeof(char));

   // Modify string, hello + user specified name
   strcpy(greetingPtr, "Hello ");
   strcat(greetingPtr, nameArr);
   strcat(greetingPtr, ".");

   // Output greeting and name
   printf("%s\n", greetingPtr);

   free(greetingPtr);

   return 0;
}
```



```
Enter name: Julia
Hello Julia.

...

Enter name: John Smith
Hello John Smith.
```

The nameArr array is used for reading the user input, so nameArr is statically allocated with a fixed size, defining a limit to the length of the name that the program can support.

In contrast, the size of the greeting is determined at runtime based upon the actual user-entered name length. `char* greetingPtr;` declares a char pointer variable named greetingPtr. Once the user has entered a name and the newline character at the end of the name string has been removed, `strlen(nameArr)` determines the number of characters within that name. In addition to the length of the string, 8 characters are needed for the greeting — 6 for the string "Hello ", 1 for the ".", and 1 for the end-of-string null character. Thus, the total number of bytes required for the greeting array can be determined using the expression `strlen(nameArr) + 8) * sizeof(char)`. For example, if the user enters the name "Julia", the total number of characters allocated for the greeting array will be 5 + 8 = 13.

The program can then create the greeting array by first copying the string "Hello " to the greeting array using the statement `strcpy(greetingPtr, "Hello ");`. The statement `strcat(greetingPtr, nameArr);` then appends the user-entered name to the end of the string stored within greetingPtr. Finally, `strcat(greetingPtr, ".");` appends a "." to the end of the string.

To create a dynamically allocated copy of a string, malloc() can be used to create a character array with a size equal to the number of characters within the source string plus one character for the null

character. The following program illustrates. As the length returned by `strlen(nameArr)` does not include the null character required at the end of a string, `strlen(nameArr) + 1` is used to determine the number of characters required for the dynamically allocated string. A common error is to allocate only strlen chars for a copied string, forgetting the + 1 needed for the null character.

Figure 8.5.4: Creating a dynamically allocated copy of a string.


```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
   char nameArr[50];      // User specified name
   char* nameCopy = NULL; // Output greeting and name

   // Prompt user to enter a name
   printf("Enter name: ");
   fgets(nameArr, 50, stdin);

   // Create dynamically allocated copy of name
   // +1 is for the null character
   nameCopy = (char*)malloc((strlen(nameArr) + 1) * sizeof(char));
   strcpy(nameCopy, nameArr);


   // Output greeting and name
   printf("Hello %s", nameCopy);

   // Deallocate memory
   free(nameCopy);

   return 0;
}
```

```
Enter name: Julia
Hello Julia

...

Enter name: John Smith
Hello John Smith
```

PARTICIPATION ACTIVITY 8.5.2: Creating and modifying strings.

1) Given an existing string sentStart, complete the following statement to allocate a new string songVerse that is large enough to store the string sentStart plus seven additional characters.

```
songVerse = (char*)malloc((
                          ) *
sizeof(char));
```


Check Show answer

2) In a single statement, copy the string pointed to by the char*

sentStart to the string pointed to by the char* songVerse.

Check Show answer

CHALLENGE ACTIVITY 8.5.1: Pointers for allocating a C string.

Use strlen(userStr) to allocate exactly enough memory for newStr to hold the string in userStr (Hint: do NOT just allocate a size of 100 chars).

347282.2052980.qx3zqy7

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(void) {
   char userStr[100];
   char* newStr = NULL;

   fgets(userStr, 100, stdin);

   /* Your solution goes here  */

   strcpy(newStr, userStr);
   printf("%s\n", newStr);

   return 0;
}
```

Run

View your last submission

8.6 The realloc function

This section has been set as optional by your instructor.

The ***realloc*** function re-allocates an original pointer's memory block to be the newly-specified size. realloc() can be used to both increase or decrease the size of dynamically allocated arrays. realloc() returns a pointer to the re-allocated memory. The pointer returned by realloc() may or may not differ from the original pointer. For example, if realloc() is used to increase a memory block but the function doesn't find enough available memory at the existing block's end, the function finds a large enough block of memory at another location in memory, and copies the existing block's contents to the new block.

In its most common form, the pointer returned from realloc() will be assigned to the same pointer provided as the input argument, using the form:

Construct 8.6.1: The realloc function.

```
pointerVariable = (type*)realloc(pointerVariable, numElements * sizeof(type))
```

The following program computes the average of several user-entered values stored within a dynamically allocated array. The array is reallocated each time the user specifies the number of integers values.

Figure 8.6.1: Dynamically reallocating the size of an array.

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
   int* userVals = NULL; // No array yet
   int numVals;
   int i;
   char userInput;
   int userValsSum;
   double userValsAvg;

   userInput = 'c';

   while (userInput == 'c') {

      printf("Enter number of integer values: ");
      scanf(" %d", &numVals);

      if (userVals == NULL) {
         userVals = (int*)malloc(numVals * sizeof(int));
      }
      else {
         userVals = (int*)realloc(userVals, numVals * sizeof(int));
      }

      printf("Enter %d integer values...\n", numVals);
      for (i = 0; i < numVals; ++i) {
         printf("Value: ");
         scanf("%d", &(userVals[i]));
      }

      // Calculate average
      userValsSum = 0;
      for (i = 0; i < numVals; ++i) {
         userValsSum = userValsSum + userVals[i];
      }
      userValsAvg = (double)userValsSum / (double)numVals;

      printf("Average = %lf\n", userValsAvg);

      printf("\nEnter 'c' to compute another average (any other key to quit): ");
      scanf(" %c", &userInput);
   }

   free(userVals);

   return 0;
}
```


©zyBooks 12/22/21 12:35 1026490
Jacob Adams
UACS100Fall2021

```
Enter number of integer values: 3
Enter 3 integer values...
Value: 13
Value: 11
Value: 17
Average = 13.666667

Enter 'c' to compute another average (any other key to quit): c
Enter number of integer values: 7
Enter 7 integer values...
Value: 10
Value: 14
Value: 56
Value: 23
Value: 18
Value: 3
Value: 6
Average = 18.571429

Enter 'c' to compute another average (any other key to quit): q
```


PARTICIPATION ACTIVITY 8.6.1: realloc.

1) Given an int* pointer dynInputVals that points to an existing dynamically allocated array of integers, complete the following statement to reallocate the array to have 14 elements.

```
dynInputVals = (int*)realloc(
               , 14 *
sizeof(int));
```

Check Show answer

2) Given a double* pointer sensorVals that points to an existing dynamically allocated array of 200 elements, complete the following statement to reallocate the size of the array to have only 2 elements.

```
sensorVals =
(double*)realloc(sensorVals,
                 );
```

Check Show answer

8.7 Memory regions: Heap/Stack

A program's memory usage typically includes four different regions:

- ***Code*** — The region where the program instructions are stored.
- ***Static memory*** — The region where global variables (variables declared outside any function) as well as static local variables (variables declared inside functions starting with the keyword "static") are allocated. Static variables are allocated once and stay in the same memory location for the duration of a program's execution.
- ***The stack*** — The region where a function's local variables are allocated during a function call. A function call adds local variables to the stack, and a return removes them, like adding and removing dishes from a pile; hence the term "stack." Because this memory is automatically allocated and deallocated, it is also called ***automatic memory***.
- ***The heap*** — The region where the malloc function allocates memory, and where the free function deallocates memory. The region is also called ***free store***

PARTICIPATION ACTIVITY 8.7.1: Use of the four memory regions.

Animation captions:

1. The code regions store program instructions. myGlobal is a global variable and is stored in the static memory region. Code and static regions last for the entire program execution.
2. Function calls push local variables on the program stack. When main() is called, the variables myInt and myPtr are added on the stack.
3. malloc allocates memory on the heap for an int and returns the address of the allocated memory, which is assigned to myPtr. free deallocates memory from the heap.
4. Calling MyFct() grows the stack, pushing the function's local variables on the stack. Those local variables are removed from the stack when the function returns.
5. When main() completes, main's local variables are removed from the stack.

PARTICIPATION ACTIVITY 8.7.2: Stack and heap definitions.

Static memory | Automatic memory | Free store | The heap | The stack | Code

A function's local variables are

	allocated in this region while a function is called.
	The memory allocation and deallocation operators affect this region.
	Global and static local variables are allocated in this region once for the duration of the program.
	Another name for "The heap" because the programmer has explicit control of this memory.
	Instructions are stored in this region.
	Another name for "The stack" because the programmer does not explicitly control this memory.

Reset

8.8 Memory leaks

Memory leak

A ***memory leak*** occurs when a program that allocates memory loses the ability to access the allocated memory, typically due to failure to properly destroy/free dynamically allocated memory. A program's leaking memory becomes unusable, much like a water pipe might have water leaking out and becoming unusable. A memory leak may cause a program to occupy more and more memory as the program runs, which slows program runtime. Even worse, a memory leak can cause the program to fail if memory becomes completely full and the program is unable to allocate additional memory.

A common error is failing to free allocated memory that is no longer used, resulting in a memory leak. Many programs that are commonly left running for long periods, like web browsers, suffer from known memory leaks — a web search for "<your-favorite-browser> memory leak" will likely result in numerous hits.

PARTICIPATION

ACTIVITY 8.8.1: Memory leak can use up all available memory.

Animation captions:

1. Memory is allocated for newVal each loop iteration, but the loop does not deallocate memory once done using newVal, resulting in a memory leak.
2. Each loop iteration allocates more memory, eventually using up all available memory and causing the program to fail.

Garbage collection

*Some programming languages, such as Java, use a mechanism called **garbage collection** wherein a program's executable includes automatic behavior that at various intervals finds all unreachable allocated memory locations (e.g., by comparing all reachable memory with all previously-allocated memory), and automatically frees such unreachable memory. Some non-standard C implementations also include garbage collection. Garbage collection can reduce the impact of memory leaks at the expense of runtime overhead. Computer scientists debate whether new programmers should learn to explicitly free memory versus letting garbage collection do the work.*

PARTICIPATION ACTIVITY 8.8.2: Memory leaks.

Garbage collection **Memory leak** **Unusable memory**

	Memory locations that have been dynamically allocated but can no longer be used by a program.
	Occurs when a program allocates memory but loses the ability to access the allocated memory.
	Automatic process of finding and freeing unreachable allocated memory locations.

Reset

©zyBooks 12/22/21 12:35 1026490
Jacob Adams
UACS100Fall2021